Intel® Extreme Graphics 2: Developer's Guide

Whitepaper

January 15, 2004
Revision 1.2

1 Source: Mercury Research Q32003 PC Graphics Report

2 Hyper-Threading (HT) Technology requires a computer system with an Intel® Pentium® 4 processor supporting HT Technology and a HT Technology enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. See www.intel.com/info/hyperthreading for more information including details on which processors support HT Technology.

Contents

1 Introduction 3

2 Intel Graphics Architecture 3

2.1 Platform Overview 3

2.1.1 Intel® 865G Chipset Platform Features 3

2.1.1 Intel® Extreme Graphics 2 Features 3

2.2 Zone Rendering 2 3

2.3 Dynamic Video Memory Technology (DVMT) 2.0 3

3 Developer Tips 3

3.1 Identification code 3

3.2 Making Pre-Runtime Decisions 3

3.2.1 Transform and Lighting Requirements 3

3.2.2 Memory Requirements 3

3.3 Enabling Zone Rendering 2 3

3.3.1 Change Render Targets Outside of Scene 3

3.3.2 Avoid Reading Back from Buffers 3

3.3.3 Avoid Locking the Buffers 3

3.3.4 Take Care When Clearing Buffers 3

3.3.5 Reduce Memory Bandwidth If Necessary 3

3.4 Use of Z-bias 3

4 Summary 3

5 Appendix A: Creating a DirectX 9 Device, Identifying Intel 865G Chipset 3

6 Appendix B: Intel Integrated Graphics Part IDs 3

7 References 3

Figures

Figure 1: 865G Chipset Platform Block Diagram 3
Figure 2: Conventional Rendering versus Zone Rendering 3
Figure 3: A sample DirectX 9 program, creating a device using software vertex processing 3
Figure 4: DirectX 7 Solution to Checking Accurate Available Memory for Extreme Graphics 2 3
Figure 5: DirectX 8 and 9 Solution to Checking Accurate Available Memory for Extreme Graphics 2 3
Figure 6: Example of Z-fighting – A Wall and Poster Share a Plane 3
Figure 7: Code Snippet Showing Alternative to Using DirectX Z-bias Call 3
Figure 8: Z-fighting Solved with Projection Modification Solution 3

Revision History

Revision Number	Description	Revision Date
1.0	Initial Publication	10/24/2003
1.2	Updated and revised • General grammatical corrections and clarifications • Clarifications to “DVMT 2.0” section • Improved 865G identification example • Improved software vertex processing device creation example • Clarifications to “Enabling Zone Rendering 2” section • Appendix A now contains working source code to identifying the 865G chipset and creating a software vertex processing device	12/31/2003

1 Introduction

Desktop systems utilizing Intel graphics continue to increase in volume and penetration into the consumer marketplace. According to a recent Mercury Research report on the PC Graphics market, Intel was the number one supplier of graphics solutions to new PC purchasers[1], resulting in a growing installed user base that offers application developers a significant market opportunity By providing graphic capabilities integrated into mainstream and value desktop computing platforms, Intel lowers the cost of PC components and allows a broader base of users to have access to high quality and solid mainstream features/performance. Each new generation of Intel's graphics products will continue to provide increasing levels of 2D, 3D, and video capability and performance.

The latest generation of Intel graphics, called Intel® Extreme Graphics 2, provides new features and significant performance improvements over previous generations by offering advanced techniques such as Zone Rendering 2 and Dynamic Video Memory Technology (DVMT) 2.0. These features are unique to Intel products and are designed to provide the required level of graphics performance needed for mainstream computing.

This paper will walk through each of these technologies at a high level while interjecting some key software development tips that are necessary to take full advantage of what Intel Extreme Graphics 2 has to offer. By utilizing this information you may see some significant improvements in your application's performance on the Intel integrated graphics architecture and be able to reach a broader base of customers for your application!

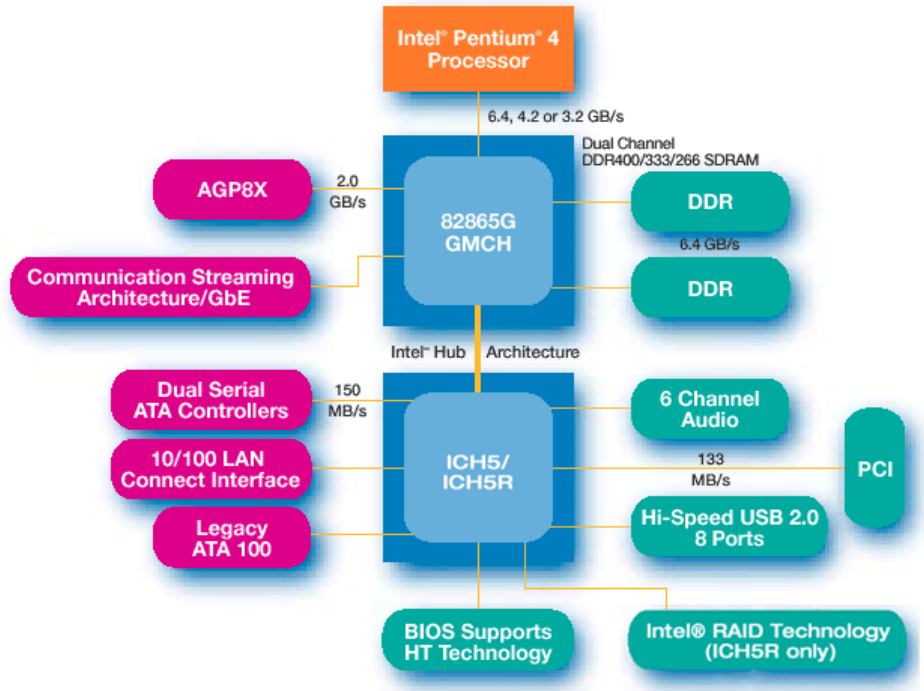
[1] Source: Mercury Research Q32003 PC Graphics Report

2 Intel Graphics Architecture

2.1 Platform Overview

Intel® Extreme Graphics 2, available in the Intel® 865G chipset, is designed to take full advantage of the power an Intel® Pentium® 4 processor brings to the PC. Graphics is just one of the many features of the Intel 865G chipset. The 865G chipset also offers support for Hyper-Threading Technology[2], an AGP 8x interface, DDR 266/333/400 MHz support, dual channel memory interface to significantly improve memory subsystem performance, AC97 audio codec support, Gigabit Ethernet support, and Hi-Speed USB 2.0*. These new features provide a powerful basis for a wide range of PC uses.

Figure 1: 865G Chipset Platform Block Diagram



2.1.1 Intel® 865G Chipset Platform Features

Table 1: Intel ® 865G Chipset Platform Features

FEATURES	BENEFITS
800/533/400 MHz System Bus	Supports platform longevity with the highest Intel processor frequencies and delivers greater system bandwidth.
Intel® Hyper-Threading Technology Support	Delivers increased system responsiveness and performance.
478-pin Processor Package Compatibility	Supports the highest performance Intel desktop processors with the flexibility to support other 478-pin Intel processors.
Intel® Extreme Graphics 2 Technology	Fourth-generation graphics architecture supports the latest APIs, allowing software developers to create real-life environments and characters.
Intel® Hub Architecture	Dedicated data paths deliver maximum bandwidth for I/O-intensive applications.
Dual-Channel DDR 400/333/266 SDRAM	Flexible memory technology allows a full spectrum of DDR usage from highest performance to more cost-effective systems.
Intel® **Digital** Video Output Interface	Two DVO ports offer maximum display (digital CRT, FP or TV) flexibility through the standard AGP connector.
AGP8X Interface	Highest bandwidth graphics interface enables upgradeability to latest graphics cards.
Integrated Hi-Speed USB 2.0	Eight ports offer up to 40x greater bandwidth over USB 1.1 for a variety of today's demanding high-speed I/O peripherals.
Dual Independent Serial ATA Controllers	Facilitates high-speed storage transfers and easy hard drive upgrades.
Intel® RAID Technology	Enables extreme storage performance for Serial ATA hard disks.
Ultra ATA/100	Takes advantage of the existing industry HDD and optical drive interfaces.
AC '97 Audio Controller	Dolby* Digital 5.1 surround sound, delivering six channels of enhanced sound quality.
Intel® Communication Streaming Architecture	Wire-speed GbE with Dedicated Network Bus for performance network connectivity.
Low-Power Sleep Mode	Saves system energy usage

2.1.1 Intel® Extreme Graphics 2 Features

2.2 Zone Rendering 2

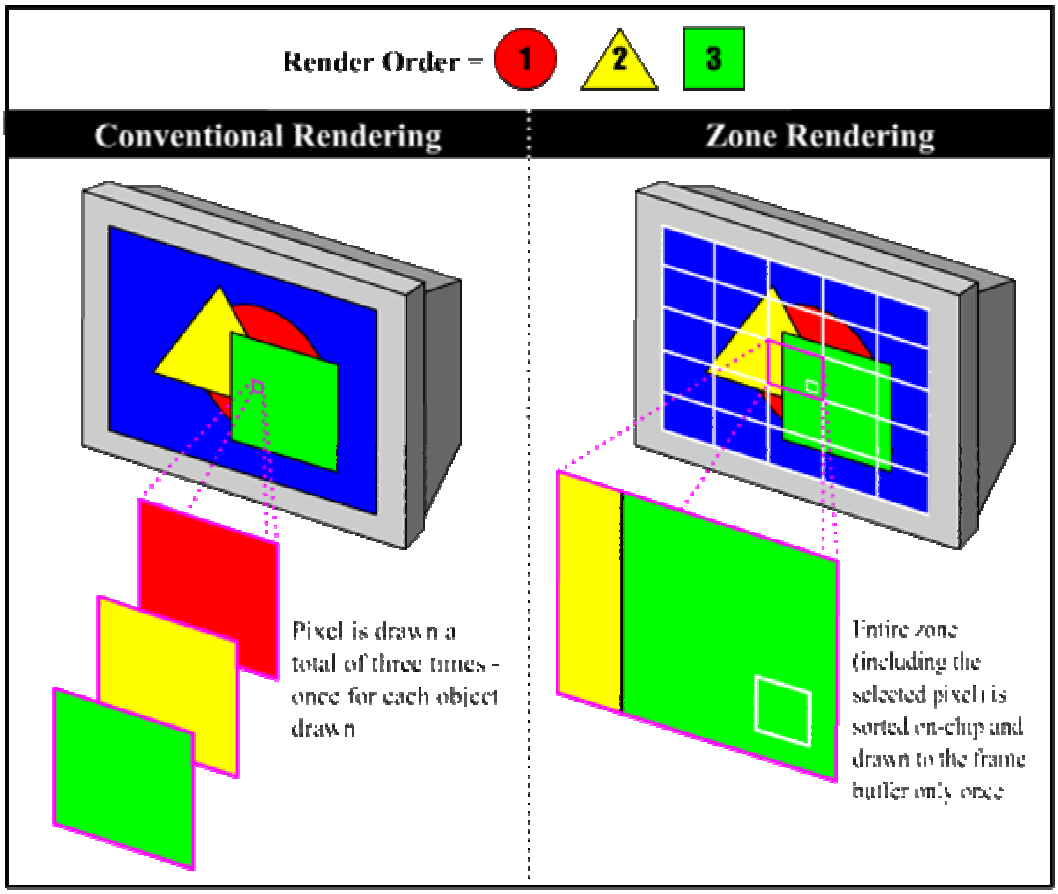
Zone Rendering 2 is a tile based rendering system designed to reduce memory bandwidth and maximize rendering performance. To understand how it works, we must first have a very basic understanding of conventional rendering.

With conventional rendering, a scene made up of various 3D models is sent to the graphics hardware where each model and its associated polygons undergo a slew of matrix multiplications to transform them from model space (the local coordinate system of each object) to world space (the coordinate system relative to the entire scene) and finally to view space (the viewer's coordinate system).

Next, light values are applied to the vertices of each triangle which are then converted into pixel or screen coordinates. Each resulting pixel is then given the proper texture color (or texel) and depth-tested (depth testing is also known as z-testing, as z represents the depth from the screen directly back in towards the monitor) to see if it will be visible or if another pixel closer to the viewer is blocking it. Since the triangles are processed in the order they are received from the hardware (in the case of the 865G chipset, they are received from the Pentium 4 processor), many pixels are written over several times as triangles closer to the viewer are placed over those further back in the scene. This redundant rendering is memory bandwidth heavy and does not provide the optimal results for the 865G chipset.

Zone rendering aims to improve the memory efficiency by reducing memory traffic. Like conventional rendering the scene is passed to hardware where the polygons it consists of are transformed into view space and their vertices lit, but rather than going directly through screen space conversion, zone rendering first sorts each polygon by their zone. Since each zone can fit in the chipset's on-chip cache the depth-testing and pixel blending operations are done quickly on-chip. This also means that each pixel is written to frame-buffer memory only once!

Figure 2: Conventional Rendering versus Zone Rendering



The amount of memory bandwidth required to render a scene with conventional rendering can be significantly more than the amount required to render a scene with Zone Rendering 2. The impact to memory bandwidth (and performance) of using Zone Rendering 2 scales, based on the depth complexity of a given scene. While there are no specific coding techniques required to enable Zone Rendering 2, there are several things a programmer can do to ensure that they take advantage of the performance benefits zone rendering has to offer. These tips are referenced in detail in Section *3.3 – Enabling Zone Rendering 2*.

2.3 Dynamic Video Memory Technology (DVMT) 2.0

Intel Extreme Graphics 2 utilizes a shared memory architecture – system memory is used for both graphics and system purposes. Instead of using dedicated local memory, as is the case on the majority of discrete graphics cards today, a portion of the system memory is allocated to be used as video memory. A small amount of system memory is permanently allocated to

video memory by the BIOS. This amount can be one, four, eight, sixteen or in some rare cases thirty-two megabytes.

DVMT 2.0 allows additional system memory to be dynamically allocated for graphics usages based on application need. Once the application is closed, the memory that was allocated is released and is then available for system use. The purpose of dynamically allocating memory for graphics use is to ensure a solid balance between system performance and graphics performance.

For example, if a user is simply editing text, there would be no need for the graphics to take up a large amount of the system's memory. In such a case, it would be best if more memory was allocated to the system. On the other hand, if the user was to start up a 3D game, there would be a need for more of the shared memory to be used as graphics memory.

On boot-up the user can choose in the system's BIOS the amount of system memory to be permanently used by the graphics controller. Once selected, this memory will never be given back to the system. This memory is also reported as local video memory in Microsoft DirectX* applications. Once the operating system is started the graphics driver will then dynamically allocate graphics memory based on requests from each application run by the user. For systems with 128 MB or less of system memory a *maximum* of 32 MB will be set aside for graphics (memory set aside by the BIOS + memory dynamically allocated by the driver). For systems with more than 128 MB of memory a *maximum* of 64 MB will be allocated for use by the graphics controller. These maximum values include both the permanently allocated memory set aside in the BIOS as well as the dynamically allocated memory.

Table 2: Maximum Video/Graphics Memory Allocated Based on Total System Memory for the 865G Chipset

Total System Memory	Maximum Video Memory
≤ 128 MB	32 MB
> 128 MB	64 MB

3 Developer Tips

3.1 Identification code

To target features specifically on Intel's Extreme Graphics 2, Device and Vendor ID information should be used to properly detect the device. This can be read from the PCI configuration space or through DirectX (version 8 or later) by using the GetAdapterIdentifier() function.

```
DWORD behaviorFlags;             // Used to describe vertex processing type
D3DADAPTER_IDENTIFIER9 adapterID; // Used to store device info

// Gather the primary adapter's information...
if(g_pd3dDevice->GetAdapterIdentifier( 0, 0, &adapterID ) != D3D_OK )
    exit(-1);

if ( ( adapterID.VendorId == 0x8086 ) && ( adapterID.DeviceId == 0x2572 ) )
{
    // 865G is current adapter...
    .
    .
    .
}
```

Table 3: 865G Chipset Vendor and Device Identification

Vendor ID	**0x8086**
Device ID	**0x2572**

See Appendix B for a list of previous Intel Integrated Graphics Part IDs.

3.2 Making Pre-Runtime Decisions

Many applications determine the quality settings that will be used based on information they get about the graphics solution. Some assumptions made about Intel's integrated graphics solutions may lead to non-optimal application settings or even prohibit an application that would otherwise operate properly from running at all. Performance assumptions based on the use of software transformation and lighting is the most common mistake and usually the most costly to fix. Incorrectly determining the amount of available graphics memory is another area that can be problematic with integrated graphics in general. The following sub-sections (3.2.1 and 3.2.2) discuss how to avoid these two common issues.

3.2.1 Transform and Lighting Requirements

A common assumption made by application developers is that hardware transformation and lighting is required to achieve a certain level of performance – the intended result is a "good experience" or no experience at all. However, the Intel's Extreme Graphics 2 engine provides a balanced graphics pipeline by allowing the Intel Pentium 4 processor to perform the transformation and lighting operations, which is often more than capable of achieving a "good experience."

In using the CPU for transform and lighting operations, the DirectX transform and lighting pipe optimized for the Pentium 4 processor is utilized. This "Processor Specific Graphics Pipeline" (PSGP) allows the Intel's Extreme Graphics controller to offload the transform and lighting operations to software while still providing excellent performance.

If application performance requirements are based on actual performance rather than on assumptions of performance based on features, this should allow for a "good experience" on many systems.

Use the Direct3D D3DCREATE_SOFTWARE_VERTEXPROCESSING definition to create a device that uses software vertex processing.

Figure 3: A sample DirectX 9 function: Detects Intel 865G Chipset and Enables Software Vertex Processing (see Appendix A for full source)

```
//-----------------------------------------------------------------------------
// Name: SetVertexProcessingMode
// Desc: Checks HW TnL caps and IDs 865G to enable SW TnL
//-----------------------------------------------------------------------------
DWORD SetVertexProcessingMode( LPDIRECT3D9 pD3D )
{
    DWORD      vertexprocessingmode;  // vertex processing mode
    D3DCAPS9   caps;                  // structure that stores device caps...
    D3DADAPTER_IDENTIFIER9 adapterID;  // Used to store device info

     // Check the capabilities...store into "caps"...
    if( g_pD3D->GetDeviceCaps( 0, D3DDEVTYPE_HAL, &caps ) != D3D_OK )
    {
        return E_FAIL; // exit if reading caps fails...
    }

    // check if hardware TnL is supported...
    if ( ( caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT ) != 0 )
    {
        vertexprocessingmode = D3DCREATE_HARDWARE_VERTEXPROCESSING;
    }
    else
    {
        // check vendor and device ID and enable software vertex processing for
        // Intel(R) 865G...

        // Gather the primary adapter's information...
```

```
        if( g_pD3D->GetAdapterIdentifier( 0, 0, &adapterID ) != D3D_OK )
        {
            return E_FAIL;
        }

        if ( ( adapterID.VendorId == 0x8086 ) &&
             ( adapterID.DeviceId == 0x2572 ) )
        {
            vertexprocessingmode = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
        }
        else
        {
            // chip does not meet requirements...
            return E_MINSPEC;
        }
    }

    return vertexprocessingmode;
}
```

3.2.2 Memory Requirements

Another check that is often done before actually executing the application is a check for the amount of available free graphics or video memory. As a result of the dynamic allocation of graphics memory performed by Intel's integrated graphics devices (based on application requests), developers need to take care in ensuring they understand all of the memory that is truly available to the graphics device. Memory checks that only supply the amount of "local" graphics memory available, do not supply an appropriate value for Intel's integrated graphics devices.

To accurately detect the amount of memory available to Intel's integrated graphics devices, developers need to check the total video memory availability. Local memory is considered to be the memory permanently set aside by the BIOS for use as graphics memory. Non-local video memory is the memory beyond the local video memory that was dynamically allocated based on requests from applications. Both local and non-local video memory combine to equal the total amount of video memory and each are handled identically for memory accesses.

The code snippet below outlines the function calls necessary to most accurately check the memory available for use by the graphics controller within DirectX.

Figure 4: DirectX 7 Solution to Checking Accurate Available Memory for Extreme Graphics 2

```
...

DDSCAPS2 ddsVidMemcaps;
ZeroMemory(&ddsVidMemcaps, sizeof(DDSCAPS2));
ddsVidMemcaps.dwCaps = DDSCAPS_VIDEOMEMORY;
```

```
hRet = g_pDD->GetAvailableVidMem(&ddsVidMemcaps, &dwVidMemTotal,
       &dwVidMemFree);

...
```

Figure 5: DirectX 8 and 9 Solution to Checking Accurate Available Memory for Extreme Graphics 2

```
...

int AvailableTextureMem = g_pd3dDevice->GetAvailableTextureMem();

...
```

3.3 Enabling Zone Rendering 2

Enabling Zone Rendering 2 technology can result in a significant increase in system performance. However, in order to maximize the benefit of Zone Rendering 2 on a given application, there are some recommendations of which developers need to be aware. In some cases, the integrated graphics device does not have the resources available to efficiently render by zone. In these cases, Intel's integrated graphics devices are forced out of Zone Rendering 2 mode into a classic rendering mode. Intel recommends that developers attempt to maintain Zone Rendering 2 whenever possible. While it is difficult to know whether or not zone rendering is enabled, there are several things developers should check if it is believed that Zone Rendering 2 is not enabled (e.g. lower than expected frame rate). The topics discussed below represent the most common reasons for Intel's integrated graphics devices to be forced out of Zone Rendering 2 mode into classic rendering mode.

3.3.1 Change Render Targets Outside of Scene

Changing rendering targets is used for creating a variety of effects. Where these calls are made can have a significant performance impact on the performance of Intel Extreme Graphics 2. It is recommended that if any changes to the render target are made, they should be done before or after the scene is rendered.

For example, if a render to texture technique is used to create a shadow effect, the render to texture could be completed at a variety of points. Completing the render to texture before the scene will result in increased performance by allowing Zone Rendering 2 mode to stay active and by avoiding unnecessary buffer evictions.

3.3.2 Avoid Reading Back from Buffers

Reading back from color, z, or stencil buffers, has a significant impact to performance on any graphics chip. The read backs themselves are slow and can potentially cause Intel's integrated graphics devices to be forced out of Zone Rendering 2 mode into classic rendering mode which further degrades performance. It is recommended to avoid reading back data from any of the buffers.

3.3.3 Avoid Locking the Buffers

Locking buffers is often done for synchronization reasons or occasionally for creating visual effects. Unfortunately, locking buffers can have a significant impact to performance and will possibly cause Intel's integrated graphics devices to be forced out of Zone Rendering 2 mode into classic rendering mode. It is recommended to lock buffers sparingly, if at all.

3.3.4 Take Care When Clearing Buffers

There are a wide variety of reasons for clearing any one of the buffers (depth, color, or stencil). How these buffers are cleared will determine whether there is a performance impact in the application.

Clearing the buffers together and once per frame is the best option. In cases when this is not possible, the depth and stencil buffers should be cleared together.

Similarly, partially clearing buffers can have a negative impact on performance. Fast clear options that allow the entire buffer to clear quickly will not be implemented if a buffer is only partially cleared.

3.3.5 Reduce Memory Bandwidth If Necessary

Zone Rendering 2 improves performance by reducing memory bandwidth. However, Intel's integrated graphics devices can be forced out of Zone Rendering 2 mode into classic rendering mode by providing too much geometry, texture, and/or state information to memory. Below is a list of potential ways to reduce the memory bandwidth required to render a scene:

- Use compressed textures
- Use D3DPOOL_MANAGED or D3DPOOL_DEFAULT when allocating surface, buffer, or texture memory
- Reduce texture size or quality
- Use level-of-detail
- Reduce the content footprint by employing efficient culling algorithms

3.4 Use of Z-bias

DirectX allows developers to resolve z-fighting issues by applying a "z-bias" to the polygons that should appear closer to the viewer. Often this is used for showing views like bullet holes or posters on walls. While applying a z-bias is an effective solution, it does not generate the same results on all graphics hardware. This can result in a lot of custom 'tweaking' of the z-bias values and subsequent testing across a wide array of hardware.

Figure 6: Example of Z-fighting – A Wall and Poster Share a Plane

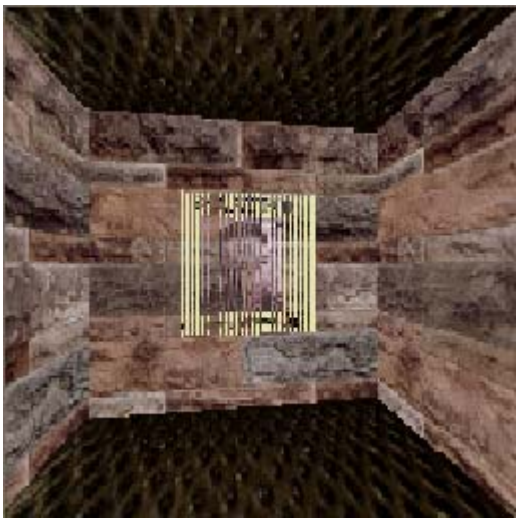


Figure 6 above demonstrates the type of visual artifact commonly seen when z-fighting occurs. Using the D3DRS_DEPTHBIAS render state in DirectX (D3DRS_ZBIAS on DirectX 8 and earlier) addresses this issue, however, the same visual artifacts can occur on other hardware.

An alternate method of addressing this issue is to load a new projection matrix in which the near and far clipping planes have been pushed out (away from the viewer). By loading this projection matrix before any objects that appear closer to the viewer, the desired object is placed closer in the z-buffer. Sample of code that accomplishes this can be seen below. In this case it is applying a z-bias to the poster, so that it is correctly displayed on the wall.

Figure 7: Code Snippet Showing Alternative to Using DirectX Z-bias Call

```
// Two projection matrices are created
D3DXMATRIX matProj, matProj_zbias;

// Original projection is created...
D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
```

```
// The "zbiased" projection has it near and far clipping planes pushed out…
D3DXMatrixPerspectiveFovLH( &matProj_zbias, D3DX_PI/4, 1.0f, 1.5f, 110.0f );

. . .

// Original projection is loaded
g_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );

// Wall is rendered...

// "zbiased" projection is loaded...
g_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj_zbias );

// Poster is rendered...

// Original projection is reloaded...
g_pd3dDevice->SetTransform( D3DTS_PROJECTION, &matProj );

. . .
```

While some adjustments to the projection matrix may still be necessary to get the desired results, this technique is more consistent across a variety of graphics hardware. Below we can see the result of the alternate solution.

Figure 8: Z-fighting Solved with Projection Modification Solution

4 Summary

Intel’s Integrated Graphics architecture is unique in the PC marketplace. It is designed to provide a more balanced use of all the platform components, not just the graphics core, to contribute to an excellent user experience for mainstream computing with low added cost to the complete system cost. The increasing use of Intel's graphics architecture in the marketplace provides incentive to support this solution if your application is targeted at mainstream PC users.

As this document described, there are some key paths you can take in your application development to provide a better user experience on Intel's graphics architecture. By implementing these examples into your application you will likely see improvements in your application's visual quality and performance on Intel integrated graphics platforms.

If you would like additional information, be sure to take a look on developer.intel.com and search on graphics. Questions, comments, and concerns can be sent to kipp.owens@intel.com.

5 Appendix A: Creating a DirectX 9 Device, Identifying Intel 865G Chipset

The program below is a very simple program that checks the device capabilities of the primary adapter and sets the vertex processing mode to software transform and lighting if it detects the Intel 865G chipset. A Direct3D* device is created and a single triangle is drawn.

The program is a modification of the DirectX 9 SDK tutorial "Vertices." It requires the DirectX 9 SDK to compile and must be linked with d3d9.lib.

```
#include <d3d9.h>
#include <string.h>

//-----------------------------------------------------------------------------
// Global variables
//-----------------------------------------------------------------------------
LPDIRECT3D9             g_pD3D       = NULL; // Used to create the D3DDevice
LPDIRECT3DDEVICE9       g_pd3dDevice = NULL; // Our rendering device
LPDIRECT3DVERTEXBUFFER9 g_pVB        = NULL; // Buffer to hold vertices
DWORD                 g_VertexProcessingMode = 0; // Used to set SW or HW vert
proc.

// A structure for our custom vertex type
struct CUSTOMVERTEX
{
    FLOAT x, y, z, rhw; // The transformed position for the vertex
    DWORD color;        // The vertex color
};

// Our custom FVF, which describes our custom vertex structure
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZRHW|D3DFVF_DIFFUSE)
#define E_MINSPEC (-3)  // Error code for parts not meeting min spec



//-----------------------------------------------------------------------------
// Name: SetVertexProcessingMode
// Desc: Checks HW TnL caps and IDs 865G to enable SW TnL
//-----------------------------------------------------------------------------
DWORD SetVertexProcessingMode( LPDIRECT3D9 pD3D )
{
   DWORD      vertexprocessingmode;  // vertex processing mode
   D3DCAPS9   caps;                  // structure that stores device caps...
   D3DADAPTER_IDENTIFIER9 adapterID;  // Used to store device info

    // Check the capabilities...store into "caps"...
   if( g_pD3D->GetDeviceCaps( 0, D3DDEVTYPE_HAL, &caps ) != D3D_OK )
   {
       return E_FAIL; // exit if reading caps fails...
   }

   // check if hardware TnL is supported...
   if ( ( caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT ) != 0 )
   {
       vertexprocessingmode = D3DCREATE_HARDWARE_VERTEXPROCESSING;
   }
   else
```

```
    {
        // check vendor and device ID and enable software vertex processing for
        // Intel(R) 865G...

        // Gather the primary adapter's information...
        if( g_pD3D->GetAdapterIdentifier( 0, 0, &adapterID ) != D3D_OK )
        {
            return E_FAIL;
        }

        if ( ( adapterID.VendorId == 0x8086 ) &&
             ( adapterID.DeviceId == 0x2572 ) )
        {
            vertexprocessingmode = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
        }
        else
        {
            // chip does not meet requirements...
            return E_MINSPEC;
        }
    }

    return vertexprocessingmode;
}



//-----------------------------------------------------------------------------
// Name: InitD3D()
// Desc: Initializes Direct3D
//-----------------------------------------------------------------------------
HRESULT InitD3D( HWND hWnd )
{
     // Create the D3D object.
     if( NULL == ( g_pD3D = Direct3DCreate9( D3D_SDK_VERSION ) ) )
         return E_FAIL;

    char mode_str[255];

    g_VertexProcessingMode = SetVertexProcessingMode( g_pD3D );

    // Set up the structure used to create the D3DDevice
     D3DPRESENT_PARAMETERS d3dpp;
     ZeroMemory( &d3dpp, sizeof(d3dpp) );
     d3dpp.Windowed = TRUE;
     d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
     d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;

     switch( g_VertexProcessingMode )
    {
        case E_FAIL:   MessageBox( hWnd, "Error identifying GPU", "Error", MB_OK
);
                       exit( E_FAIL );

        case E_MINSPEC:    MessageBox( hWnd, "GPU does not meet minimum specs:
Intel(R) 865G or Hardware T&L chip required", "Error", MB_OK );
                       exit( E_MINSPEC );

        case D3DCREATE_HARDWARE_VERTEXPROCESSING:
                       strcpy( mode_str, "Hardware T&L Enabled" );
                       break;
        case D3DCREATE_SOFTWARE_VERTEXPROCESSING:
                       strcpy( mode_str, "Software T&L Enabled" );
                       break;
    }

    if( FAILED( g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                                      g_VertexProcessingMode, &d3dpp, &g_pd3dDevice )
) )
    {
        return E_FAIL;
```

```
    }

    // Device state would normally be set here
    MessageBox( hWnd, mode_str, "Vertex Processing Mode", MB_OK );
    return S_OK;
}



//-----------------------------------------------------------------------------
// Name: InitVB()
// Desc: Creates a vertex buffer and fills it with our vertices. The vertex
//       buffer is basically just a chuck of memory that holds vertices. After
//       creating it, we must Lock()/Unlock() it to fill it. For indices, D3D
//       also uses index buffers. The special thing about vertex and index
//       buffers is that they can be created in device memory, allowing some
//       cards to process them in hardware, resulting in a dramatic
//       performance gain.
//-----------------------------------------------------------------------------
HRESULT InitVB()
{
    // Initialize three vertices for rendering a triangle
    CUSTOMVERTEX vertices[] =
    {
        { 150.0f,  50.0f, 0.5f, 1.0f, 0xffff0000, }, // x, y, z, rhw, color
        { 250.0f, 250.0f, 0.5f, 1.0f, 0xff00ff00, },
        {  50.0f, 250.0f, 0.5f, 1.0f, 0xff00ffff, },
    };

    // Create the vertex buffer. Here we are allocating enough memory
    // (from the default pool) to hold all our 3 custom vertices. We also
    // specify the FVF, so the vertex buffer knows what data it contains.
    if( FAILED( g_pd3dDevice->CreateVertexBuffer( 3*sizeof(CUSTOMVERTEX),
                                                  g_VertexProcessingMode,
D3DFVF_CUSTOMVERTEX,
                                                  D3DPOOL_DEFAULT, &g_pVB, NULL )
) )
    {
        return E_FAIL;
    }

    // Now we fill the vertex buffer. To do this, we need to Lock() the VB to
    // gain access to the vertices. This mechanism is required becuase vertex
    // buffers may be in device memory.
    VOID* pVertices;
    if( FAILED( g_pVB->Lock( 0, sizeof(vertices), (void**)&pVertices, 0 ) ) )
        return E_FAIL;
    memcpy( pVertices, vertices, sizeof(vertices) );
    g_pVB->Unlock();

    return S_OK;
}



//-----------------------------------------------------------------------------
// Name: Cleanup()
// Desc: Releases all previously initialized objects
//-----------------------------------------------------------------------------
VOID Cleanup()
{
    if( g_pVB != NULL )
        g_pVB->Release();

    if( g_pd3dDevice != NULL )
        g_pd3dDevice->Release();

    if( g_pD3D != NULL )
        g_pD3D->Release();
}
```

```
//-----------------------------------------------------------------------------
// Name: Render()
// Desc: Draws the scene
//-----------------------------------------------------------------------------
VOID Render()
{
    // Clear the backbuffer to a blue color
    g_pd3dDevice->Clear( 0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,0), 1.0f, 0
);

    // Begin the scene
    if( SUCCEEDED( g_pd3dDevice->BeginScene() ) )
    {
        // Draw the triangles in the vertex buffer. This is broken into a few
        // steps. We are passing the vertices down a "stream", so first we need
        // to specify the source of that stream, which is our vertex buffer. Then
        // we need to let D3D know what vertex shader to use. Full, custom vertex
        // shaders are an advanced topic, but in most cases the vertex shader is
        // just the FVF, so that D3D knows what type of vertices we are dealing
        // with. Finally, we call DrawPrimitive() which does the actual rendering
        // of our geometry (in this case, just one triangle).
        g_pd3dDevice->SetStreamSource( 0, g_pVB, 0, sizeof(CUSTOMVERTEX) );
        g_pd3dDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
        g_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );

        // End the scene
        g_pd3dDevice->EndScene();
    }

    // Present the backbuffer contents to the display
    g_pd3dDevice->Present( NULL, NULL, NULL, NULL );
}



//-----------------------------------------------------------------------------
// Name: MsgProc()
// Desc: The window's message handler
//-----------------------------------------------------------------------------
LRESULT WINAPI MsgProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch( msg )
    {
        case WM_DESTROY:
            Cleanup();
            PostQuitMessage( 0 );
            return 0;
    }

    return DefWindowProc( hWnd, msg, wParam, lParam );
}



//-----------------------------------------------------------------------------
// Name: WinMain()
// Desc: The application's entry point
//-----------------------------------------------------------------------------
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR, INT )
{
    // Register the window class
    WNDCLASSEX wc = { sizeof(WNDCLASSEX), CS_CLASSDC, MsgProc, 0L, 0L,
                      GetModuleHandle(NULL), NULL, NULL, NULL, NULL,
                      "Enabling Software T&L for 865G", NULL };
    RegisterClassEx( &wc );

    // Create the application's window
    HWND hWnd = CreateWindow( "Enabling Software T&L for 865G",
                              "Intel(R) 865G Detection/Initialization",
```

```
                            WS_OVERLAPPEDWINDOW, 100, 100, 300, 300,
                              GetDesktopWindow(), NULL, wc.hInstance, NULL );

    // Initialize Direct3D
    if( SUCCEEDED( InitD3D( hWnd ) ) )
    {
        // Create the vertex buffer
        if( SUCCEEDED( InitVB() ) )
        {
            // Show the window
            ShowWindow( hWnd, SW_SHOWDEFAULT );
            UpdateWindow( hWnd );

            // Enter the message loop
            MSG msg;
            ZeroMemory( &msg, sizeof(msg) );
            while( msg.message!=WM_QUIT )
            {
                if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
                {
                    TranslateMessage( &msg );
                    DispatchMessage( &msg );
                }
                else
                    Render();
            }
        }
    }

    UnregisterClass( "Enabling Software T&L for 865G", wc.hInstance );
    return 0;
}
```

6 Appendix B – Intel Integrated Graphics Part IDs

Part	Vendor ID	Device ID
Intel® 865G Chipset	0x8086h	0x2572h
Intel® 855GM Chipset	0x8086h	0x3582h
Intel® 845G Chipset	0x8086h	0x2562h
Intel® 830M Chipset	0x8086h	0x3577h

7 References

Intel Extreme Graphics 2 Homepage: http://developer.intel.com/design/graphics2/

DVMT 2.0 Whitepaper: http://www.intel.com/design/graphics2/dvmt.htm

Zone Rendering 2 Whitepaper: http://www.intel.com/design/graphics2/zr.htm

Intel 865G Chipset Homepage: http://www.intel.com/design/chipsets/865g/index.htm

Intel 865G Chipset Datasheet: http://www.intel.com/design/chipsets/datashts/252514.htm